Semantic Foundations for Embedding HOL in Nuprl

Douglas J. Howe

Bell Labs
600 Mountain Ave., Room 2B-438
Murray Hill, NJ 07974, USA.

Abstract. We give a new semantics for Nuprl's constructive type theory that justifies a useful embedding of the logic of the HOL theorem prover inside Nuprl. The embedding gives Nuprl effective access to most of the large body of formalized mathematics that the HOL community has amassed over the last decade. The new semantics is dramatically simpler than the old, and gives a novel and general way of adding set-theoretic equivalence classes to untyped functional programming languages.

1 Introduction

Nuprl [5] and HOL [9] are interactive theorem proving systems with a number of similarities: their logics are higher-order type theories, their approaches to automated reasoning are based on that of LCF [8], and their main application has been to formal reasoning about computation. However, the two logics are very different in a number of ways. Nuprl has a constructive type theory, based on a type theory of Martin-Löf[16]. The theory contains a programming language, and all objects have a computational interpretation. Programs are reasoned about directly in logic, and the constructivity of the theory means that programs can be synthesised from proofs. On the other hand, HOL's theory is classical, and the way mathematics is encoded is similar to the way ordinary mathematics is done in ZF set theory. Functions are built in, but all other objects, such as integers and lists, are given set-theory-like encodings with the aid of the "select operator" $@x{\in}T.\ P(x)$, which denotes some x of type T such that $P(x)$.

The HOL theory has proven to be well suited to formalizing much of the mathematics of computation. The system has attracted a large number of users (almost certainly more than any theorem-proving system), and a great deal of the effort in the HOL community has gone into building the libraries of formal mathematics needed for verifying hardware and software of practical interest. A good picture of the scope and extent of this work can be obtained from the proceedings of recent meetings of the annual HOL conference (for example, [2]).

Nuprl's type theory offers a number of advantages over HOL's logic.

- *Expressive power of the type system.* Nuprl has subtypes and dependent function types. Also, through the use of universes and "sigma" types, one can express modules of the kind found in Standard ML [18].

- *Constructivity.* Experience with Nuprl has shown that for the mathematics of ordinary programs, constructivity comes at essentially no cost. Thus it seems to be a strict loss that one cannot extract programs from formal proofs in HOL.
- *Writing programs.* Nuprl includes a programming language which, while primitive, includes many of the features, such as function definition by general recursion, of a conventional functional programming language.

Most of these features have been recognized in the HOL community as desirable for HOL. See, for example, [17, 15, 20].

There have been a number of substantial applications of Nuprl (see [14] for a recent example), but there has been nothing like the sustained effort of the HOL community in formalizing mathematics useful for verification.

There are two main motivations for the present work. The first is to make up for Nuprl's relative lack of libraries of mathematics. The proposed solution is to reconcile the semantics of the two logics so that most of the mathematics developed in HOL can be directly imported into Nuprl. The goal here is a practical one, to be able to effectively use HOL mathematics in Nuprl proofs. Just about any theorem-prover can embed the logic of any other simply by formalizing the syntax of proofs, but this is not effective. We need a strong connection between the mathematics developed in Nuprl and the mathematics imported from HOL, so that HOL facts will be applicable in Nuprl proofs, and furthermore will be applicable in such a way that Nuprl's automated reasoning programs can readily incorporate them. We also need to be careful not to let HOL's classical nature spill over and destroy the constructivity of Nuprl proofs.

All the work in reconciling the semantics is on the Nuprl side. We give a new semantics of Nuprl which combines set theory with the operational semantics of Nuprl's programming language. In this semantics one can find standard models of HOL's type theory.

The other main motivation for this work is to fix a long-standing and serious problem with the semantics of Nuprl's logic. The problem is the complexity of the semantics. The semantics is operationally derived: one starts with an untyped programming language presented as a set of terms together with an evaluation relation, and then inductively builds a type system. A type system is a partial function from terms to partial equivalence relations over terms ("PERS"), and the terms in the domain of the function are called types. Thus the meaning of a type is a set of terms together with an equivalence relation over the set.

Deriving the semantics of types from an operational semantics of an untyped programming language has several advantages. One is that the approach is fairly generic. For example, there is no difficulty in substituting a language like the functional part of Standard ML (ignoring ML's types) for Nuprl's current language. Another advantage is the flexibility and expressive power of the type system.

The main cost of this approach has been the use of PERs. The PER type system itself is not particularly complicated — the difficulties arise when one extends the semantics to sequents, or "hypothetical judgements" to use Martin-

Löf's terminology. Consider, for example, the rule

$$\frac{\Gamma, x:T_1 \vdash b \in T_2 \quad \Gamma \vdash T_1 \in U_1}{\Gamma \vdash \lambda x.\, b \in T_1 \rightarrow T_2}$$

Ignoring the second premise and the list Γ of typing assumptions, a naive reading of the rule would say that in order to show that $\lambda x.\, b$ has type $T_1 \rightarrow T_2$, it suffices to assume x has type T_1 and prove that b has type T_2. But this is not sufficient. In order for $\lambda x.\, b$ to have type $T_1 \rightarrow T_2$, it must map equal members of T_1 to equal members of T_2. Thus, for this rule to be valid, the truth of the first premise must guarantee this "functionality" of $\lambda x.\, b$.

This gives rise to a "functionality semantics" for sequents. A functionality semantics is given by Martin-Löf in [16]. This semantics is itself fairly complicated. However, for technical reasons having to do with several essential practical considerations, including reasoning about general recursive programs, and collapsing Martin-Löf's four forms on judgement into one, Nuprl requires a more refined notion of functionality. Chapter 8 of the Nuprl book [5] gives a sketch of this semantics. For a better idea of the complications involved, see Allen's PhD thesis [3]. Because of this complexity, many of the existing Nuprl rules have not been completely verified, and there is a strong barrier to extending or modifying the theory. In particular, it has been a barrier to changing Nuprl to have a programming language more like SML.

The new semantics completely does away with PERs and functionality. Instead, types are simply sets of terms, and sequents essentially have the naive semantics: the sequent

$$x_1:A_1, \ldots, x_n:A_n \vdash t \in T$$

will be true if t is a member of T whenever the x_i are terms such that x_i is a member of A_i for $1 \leq i \leq n$.

Section 2 gives the technical core of the paper. In it, we show how to add objects of set theory to the operational semantics of a programming language. These objects come from a universe V of sets, whose members include functions (represented as graphs), pairs, and so on, as well as sets of these objects and equivalence classes over these sets. The approach is to extend the evaluation relation of the programming language with rules for objects in V. The technical work is to make this coherent. Because of space considerations, Section 2 does not deal with all of Nuprl's (rather large) language.

The hardest part of making this semantics work is dealing with equivalence classes. These are included to account for Nuprl's quotient type, which is essential for implementing abstract data types in Nuprl. In Nuprl, an ADT is represented by a sigma type, each member of which is a tuple consisting of an implementation type together with implementations of the operators of the ADT. The operators must satisfy the equations of the ADT. Often a desired implementation type does not have the right equality. Consider, for example, the implementation of rational numbers as pairs of integers. In this case, a quotient must be used to give the implementation type the right equality.

A quotient type in Nuprl has the form $(x, y) : A//E$, where x and y bind in E. E represents an equivalence relation over the type A. In the PER model, this is easily explained. It is simply the type that has the same members as A, but whose equivalence relation is defined by E. In the new semantics, the type will contain equivalence classes, in the usual set theoretic sense, formed from A. Since we also want the quotient to be computationally meaningful, the type will also contain "polymorphic equivalence classes" that can be computed with.

A point worth emphasizing about the new semantics is that is not particular to any programming language. When constructing V, one needs to know what the possible forms of data values are (and the current construction covers most of the forms in existing programming languages), but almost all of the technical development is independent of the rest of the language. The approach to operational semantics builds on our work described in [12]. Evidence for the robustness of this approach with respect to changes in the programming language can be found, for example, in the adaptations of our approach by Pitts and Gordon, described in [19, 7].

In Section 3, we show how to apply this semantics to the Nuprl logic as described in [5]. This is done by adding, to the programming language, operators representing Nuprl's type constructors, together with rules specifying how to "evaluate" instances of these constructors to get a member of V representing the set of all members of a type. We have to make a small change to the Nuprl rules to accommodate the new semantics. In particular, the rules for the quotient type need to incorporate the new constructor for values of the type. Also, we need to slight modify the extensionality rule (which reduces proving $f \in A \rightarrow B$ to proving $f(x) \in B$ for all $x \in A$). These rule changes have no significant practical import for Nuprl. It should be easy to adapt old proofs to use the new rules.

In Section 4 we sketch how to use this new semantics to justify an embedding of HOL. We are currently in the process of actually using this embedding. The connection between HOL and Nuprl has been implemented, and we have begun the importation of HOL theories. The immediate goal is to use these theories in a project to use Nuprl to verify the SCI cache-coherency protocol [1]. Details on the embedding of HOL and its practical applications will be the subject of a future paper.

In the last section we discuss some related work and discuss some extensions of Nuprl justified by our semantics. The appendix gives a proof sketch postponed from the body of the paper.

2 Semantics

The semantics has an operational flavour. We start with the standard cumulative hierarchy of set theory. We modify encodings of objects like functions, equivalence classes and the sets that will be used to stand for types in the type theory, so that they are distinguishable via "tags". We then remove certain ill-behaved sets, calling the resulting universe of sets V.

We then construct a "programming" language based on V and the terms of Nuprl. The semantics of this language is given as a set of rules inductively defining an evaluation relation $\Downarrow$. These rules explain how to evaluate, for example, the application of a set theoretic function, represented as a graph, to an arbitrary term. We then define an operational preorder, $\leq$, for the resulting language. Intuitively, $e \leq e'$ if e approximates e'. In particular, if $e \leq e'$, and if $C[\cdot]$ is a program context (i.e. a term with a hole in it) such that $C[e]$ evaluates to an atomic value v (an integer, say), then $C[e']$ also evaluates to v. Some examples are given in Section 2.2.

The operational preorder will be used to give set theoretic meanings to terms. If $\alpha \in V$ and $\alpha \leq a$ then α will be a possible set-theoretic meaning of a. For some a there will be many possible values of α. However, because of the removal of "ill-behaved" objects, if $\gamma \in V$ stands for a type, then for all e there will be at most one $\alpha \in \gamma$ such that $\alpha \leq e$. Thus, relative to a given type γ, terms will have unique set theoretic meanings. Furthermore, for any given e there will be at most one $\gamma \leq e$. Thus a term e will represent at most one type γ.

2.1 The Set Theoretic Universe V

We first define Z to be a large chunk of the usual cumulative hierarchy of ZF set theory. In particular, define sets Z_σ, indexed by ordinals σ, by $Z_{\sigma+1} = Pow(Z_\sigma)$, where $Pow(X)$ is the power set of X, and $Z_\tau = \bigcup_{\sigma<\tau} Z_\sigma$ if τ is a limit ordinal. Now fix some ordinal σ_0, and let $Z = \cup_{\tau<\sigma_0} Z_\tau$. For $\alpha \in Z$, define the *rank* of α, denoted $rank(\alpha)$, to be the least ordinal $\tau < \sigma_0$ such that $\alpha \in Z_\tau$.

We now define $W \subset Z$ to be elements of Z that are tagged according to a certain scheme. Let I be some set. Pick distinct sets fn, set, eq and c_i, $i \in I$, and let (a, b), for $a, b \in W$, be the standard encoding of pairs in set theory. Inductively define $W \subset Z$ as follows.

1. $(set, \gamma) \in W$ if $\gamma \subset W$.
2. $(fn, \phi) \in W$ if $\phi \subset W \times W$ and for all $(x, y), (x', y') \in \phi$, $x = x'$ implies $y = y'$.
3. $(c_i, (x_1, \ldots, x_n)) \in W$ if $x_i \in W$ for all i.
4. $(eq, \xi) \in W$ if $\xi \subset W$.

We will usually identify (fn, ϕ) and ϕ, (eq, ξ) and ξ, (set, γ) and γ, and write $c_i(\overline{x})$ for $(c_i, \overline{x})$. We use the letters ϕ, γ, and ξ exclusively for objects introduced by clauses *1*, *2* and *4* above, respectively. We use the letters α and β for arbitrary members of W.

An object $c_i(\overline{e})$ is intended to represent a value built with the data constructor c_i; ϕ is intended to represent a set-theoretic function; ξ, an equivalence class; and γ, the collection of set-theoretic meanings of a type in the type theory.

The definition of $V \subset W$ is rather technical, and was chosen to meet two requirements. One requirement is the unique-meaning property described above. The other is that V be closed under the set constructors, such as generalized cartesian product, that correspond to Nuprl's type constructors.

$$\frac{f \Downarrow \phi \quad (\alpha,\beta) \in \phi \quad \alpha \lhd a}{f(a) \Downarrow \hat{\beta}}\,(ap_\phi) \qquad \frac{f \Downarrow \lambda x.\, b \quad b[a/x] \Downarrow v}{f(a) \Downarrow v}\,(ap_\lambda)$$

$$\frac{}{\hat{\alpha} \Downarrow \hat{\alpha}}\,(\alpha) \qquad \frac{}{\lambda x.\, b \Downarrow \lambda x.\, b}\,(\lambda) \qquad \frac{}{[e] \Downarrow [e]}\,(eq) \qquad \frac{}{c_i(\overline{e}) \Downarrow c_i(\overline{e})}\,(c_i)$$

$$\frac{a \Downarrow \xi \quad \forall\, \alpha \in \xi.\ \beta \lhd f(\hat{\alpha})}{f \cdot a \Downarrow \hat{\beta}}\,(ap_\xi) \qquad \frac{a \Downarrow [a_0] \quad f(a_0) \Downarrow v}{f \cdot a \Downarrow v}\,(ap_{eq})$$

Fig. 1. Evaluation rules.

To define V, we first need to introduce a notion of consistency between members of W. It will turn out that two members that are *not* consistent cannot approximate the same term.

Definition 1. Two elements $x, y \in W$ are *consistent* if $x \uparrow y$, where $\uparrow \subset W \times W$ is defined by rank induction as follows.

- $\gamma \uparrow \gamma$.
- $\phi_1 \uparrow \phi_2$ if for all $(\alpha_1, \beta_1) \in \phi_1$ and $(\alpha_2, \beta_2) \in \phi_2$, if $\alpha_1 \uparrow \alpha_2$ then $\beta_1 \uparrow \beta_2$.
- $c_i(x_1, \ldots, x_n) \uparrow c_i(x'_1, \ldots, x'_n)$ if for all j, $x_j \uparrow x'_j$.
- $\xi \uparrow \xi'$ if for some $\alpha \in \xi$ and $\alpha' \in \xi'$, $\alpha \uparrow \alpha'$.

Definition 2. Define $V \subset W$ by rank induction as follows.

- $\gamma \in V$ if $\gamma \subset V$ and for all $\alpha, \alpha' \in \gamma$, $\alpha \uparrow \alpha'$ implies $\alpha = \alpha'$.
- $\phi \in V$ if $\phi \subset V \times V$ and $\phi \uparrow \phi$.
- $c_i(x_1, \ldots, x_n) \in V$ if $x_j \in V$ for all j.
- $\xi \in V$ if $\xi \subset V$.

Note that $\alpha \uparrow \alpha$ for all $\alpha \in V$. Hencefore all uses of the letters $\alpha, \beta, \gamma, \phi$ and ξ will be restricted to V.

2.2 A Programming Language

In this section we give a "programming" language that combines V with Nuprl's term language. We are taking Nuprl's language to include two new operators, one for constructing members of quotient types, and one for destructing them. The changes in the Nuprl rules needed to accommodate these new operators will be discussed in Section 3.

The operators in Nuprl's term language are either canonical, and are used to construct values, or are non-canonical, and are used to build terms that require evaluation to obtain a value. For the new semantics, we reclassify Nuprl's operators for building types from canonical to non-canonical (this will be expanded on in the next section). We treat Nuprl's canonical operators generically, and omit

$$\frac{e \Downarrow v \quad \alpha \lhd v}{\alpha \lhd e}\,(\Downarrow\lhd) \qquad \frac{}{\alpha \lhd \alpha}\,(\alpha\lhd) \qquad \frac{\alpha \in \xi \quad \alpha \lhd a}{\xi \lhd [a]}\,(\xi\lhd)$$

$$\frac{\forall\,(\alpha,\beta)\in\phi.\ \ \beta \lhd b[\hat{\alpha}/x]}{\phi \lhd \lambda x.\ b}\,(\phi\lhd) \qquad \frac{\forall\, j.\ \ \alpha_j \lhd e_j}{c_i(\overline{\alpha}) \lhd c_i(\overline{e})}\,(c_i\lhd)$$

Fig. 2. Approximation rules.

here all of its non-canonical operators except for function application. Extending the proofs to deal with the omitted operations is completely straightforward. In [12] we show how to define a general rule schema such that Theorem 6 holds whenever the underlying evaluation rules fit the schema, as is the case with the rules for the omitted operators.

The index set I used in the definition of V is chosen so that the c_i's can be put in one-to-one correspondence with the set of all canonical operators of Nuprl except for λ (which is the only canonical binding operator).

We build the set T of terms of our programming language by starting with an infinite set of variables and all $\gamma, \phi, \xi \in V$ as constants, and then closing under the following two rules. If f and a are terms, then $f(a)$, $f \cdot a$ and $[a]$ are terms. If $\overline{e}$ is a tuple of terms, then for all $i \in I$, $c_i(\overline{e})$ is a term. If b is a term and x is a variable, then $\lambda x.\ b$ is a term.

The usual definitions of substitution, closed term, and so on, apply to this language. Let T_0 be the set of closed terms. Inductively define an injection i from V to T_0 as follows.

$$i[\alpha] = \begin{cases} c_i(i[\alpha_1], \ldots, i[\alpha_n]) & \text{if } e \text{ has the form } c_i(\alpha_1, \ldots, \alpha_n) \\ \alpha & \text{otherwise} \end{cases}$$

We will usually write $\hat{\alpha}$ for $i[\alpha]$.

We now give a form of operational semantics for this language by giving a set of inductive rules that simultaneously define binary relations $\lhd \subset V \times T_0$ and $\Downarrow \subset T_0 \times T_0$. $\Downarrow$ will be the evaluation relation of the language, and $\lhd$ will turn out to be a restriction of the operational preorder based on $\Downarrow$.

The rules for $\Downarrow$ and $\lhd$ are given in Figures 1 and 2. We first give the intuitive meanings of these rules, and then illustrate with a few examples. Consider first the evaluation rules (Figure 1). The first two rules are for evaluation of function applications. To evaluate a term $f(a)$, one first evaluates f. If the value is an abstraction $\lambda x.\ b$, then the value is the value of $b[a/x]$ (if any). If it is a constant $\phi \in V_{fn}$, then find an ordered pair $(\alpha, \beta) \in \phi$ such that α approximates a, and return $\hat{\beta}$.

The second line of rules in Figure 1 simply says that any expression built with a value constructor evaluates to itself. The last two rules are for computing with equivalence classes. The idea is that one computes with an equivalence class by computing with its members. To evaluate $f \cdot a$, first evaluate a. If the value

is the "polymorphic" equivalence class $[a_0]$, then the result is simply the value of the function application $f(a_0)$. $[a_0]$ can be thought of as standing for any equivalence class that has a_0 as a member.

Rule (ap_ξ), covering the case where the value of f is the equivalence class ξ, is crucial. It is the reason why we can dispense with functionality in our semantics. Intuitively, the rule will force any function computing with equivalence classes to do a "run-time" check that it respects the equality represented by the classes. In particular, we want to force f to check that it returns the same value no matter what member of ξ it is applied to. Unfortunately, there is no appropriate global notion of "same value". So, this rule "guesses" a value $\beta \in V$ such that β approximates $f(\hat{\alpha})$ for all $\alpha \in \xi$, and returns $\hat{\beta}$.

Now consider the rules in Figure 2. Rule $(\xi \lhd)$ says that an equivalence class ξ approximates a polymorphic equivalence class $[a]$ if some member of ξ approximates a. Rule $(\phi \lhd)$ says that a graph ϕ approximates an abstraction $\lambda x.\ b$ if every α in the domain of ϕ, the value of ϕ at α approximates $b[\hat{\alpha}/x]$.

We now look at a few examples. In the full language, some of the c_i correspond to the integers, and there are non-canonical operators for addition etc. Let $\phi = \{(0,4),(1,5)\}$, $\phi' = \{(0,2)\}$ and $\psi = \{(\phi,17),(\phi',18)\}$. We have

- $\hat{\phi}(0+0) \Downarrow 4$ because $0 \lhd 0+0$.
- $\hat{\phi} \lhd \lambda x.\ x+4$, but not $\hat{\phi'} \lhd \lambda x.\ x+4$.
- $\hat{\psi}(\lambda x.\ x+4) \Downarrow 17$.

We now consider an example involving the quotient type. Let $\xi_1 = \{0,2,\ldots\}$ and $\xi_2 = \{1,3,\ldots\}$ (again ignoring tags). The type $(x,y) : N//even(x-y)$ will have as members $\hat{\xi_1}, \hat{\xi_2}$ and $[\underline{n}]$ for $n \geq 0$. We have $\hat{\xi_1} \lhd [2]$ but not $\hat{\xi_2} \lhd [2]$. Also, if

$$f = \lambda x.\ \mathit{if\ evenp(x)\ then\ 0\ else\ 1}$$

then $f \cdot [2] \Downarrow 0$ and $f \cdot \xi_2 \Downarrow \hat{1}$.

The evaluation relation $\Downarrow$ is idempotent, in the sense that if $e \Downarrow v$ then $v \Downarrow v$. We use the letters u and v exclusively for *values*, which are terms u such that $u \Downarrow u$. Note that, because of the rule (ap_ξ), $\Downarrow$ is not determinate: there is a term e and distinct v, v' such that $e \Downarrow v$ and $e \Downarrow v'$. However, Theorem 9 below says that this indeterminacy is inessential.

2.3 Operational Preorder and Meaning of Programs

All of the terms considered so far, except for λ-abstractions, can be written as $\tau(\overline{e})$ where τ is an operator and $\overline{e}$ is a (possibly empty) sequence of terms. Applications $f(a)$ can be thought of as having the form $ap(f,a)$. For η a binary relation on terms, define $e\ \langle\eta\rangle\ e'$ if $e = e' = x$, or if $e = \lambda x.\ b$, $e' = \lambda x.\ b'$ and $b\ \eta\ b'$, or if $e = \tau(e_1,\ldots,e_n)$, $e' = \tau(e'_1,\ldots,e'_n)$ and $e_i\ \eta\ e'_i$ for $1 \leq i \leq n$. Some of the Nuprl operators that we are omitting from the present account are binding operators. In the full account, the definition of $\langle\eta\rangle$ is extended to these operators in the obvious way.

We extend a relation $\eta \subset T_0 \times T_0$ on closed terms to a relation η° on open terms by defining $e\ \eta^\circ\ e'$ if $\sigma(e)\ \eta\ \sigma(e')$ for all substitutions σ such that $\sigma(e)$ and $\sigma(e')$ are closed.

The operational preorder is now defined as follows.

Definition 3. Let $\eta \subset T_0 \times T_0$. Define $[\eta] \subset T_0 \times T_0$ by $e\ [\eta]\ e'$ if $e \Downarrow u$ implies there exists u' such that $e' \Downarrow u'$ and one of the following holds.

1. $u\ \langle \eta^\circ \rangle\ u'$.
2. $u = \phi$, $u' = \lambda x.\, b'$ and for all $(\alpha, \beta) \in \phi$, $\hat{\beta}\ \eta\ b'[\hat{\alpha}/x]$.
3. $u = \xi$, $u' = [a']$ and for some $\alpha \in \xi$, $\hat{\alpha}\ \eta\ a'$.

Note that the mapping $\eta \mapsto [\eta]$ is monotone with respect to inclusion of relations. This allows us to make the following definition.

Definition 4. Define $\leq$ to be the largest relation $\eta \subset T_0 \times T_0$ such that $\eta \subset [\eta]$. Define $\sim$ to be the symmetric closure of $\leq$.

It is easy to show that the fixed-point equation $\leq\ = [\leq]$ holds. We will use this equation frequently (and implicitly, usually) in the rest of the paper.

As in [10, 11], it is straightforward to show that $\leq$ is a preorder (i.e. it is reflexive and transitive). This can be done by the principle of *coinduction*, which says that to prove $\eta \subset \leq$ it suffices to prove $\eta \subset [\eta]$.

Lemma 5. *For all $\alpha \in V$ and all closed terms e, $\alpha \lhd e$ if and only if $\hat{\alpha} \leq e$.*

Proof. The proof is a straightforward induction on the rank of α.

In what follows we will use $\alpha \lhd e$ and $\hat{\alpha} \leq e$ interchangeably.

The proof of the following theorem is too long to be included here. The details are not particularly interesting. The appendix gives a sketch of the proof.

Theorem 6. $\leq^\circ$ *is a* precongruence*: for all terms e and e', if $e\ \langle \leq^\circ \rangle\ e'$ then $e \leq^\circ e'$.*

An immediate consequence Theorem 6 is a substitutivity property: if $e \leq^\circ e'$ and $a \leq^\circ a'$ then $e[a/x] \leq^\circ e'[a'/x]$.

Proving the coherence theorem (Theorem 9 below) is straightforward because if a term e evaluates to both v and v', then v and v' are the same up to consistent constants. This is made precise in the following definition and lemma.

Definition 7. Define $e \bowtie e'$ if there is a term c with free variables $x_1, \ldots, x_n$ and some $\alpha_1, \ldots, \alpha_n \in V$ and $\alpha'_1, \ldots, \alpha'_n \in V$, such that $e = c[\overline{\alpha}/\overline{x}]$, $e' = c[\overline{\alpha}'/\overline{x}]$, and for each i, $1 \leq i \leq n$, $\alpha_i \uparrow \alpha'_i$.

Lemma 8. *1. If $e \bowtie e'$, $e \Downarrow v$ and $e' \Downarrow v'$, then $v \bowtie v'$.*
2. If $e \bowtie e'$, $\alpha \lhd e$ and $\alpha' \lhd e'$ then $\alpha \uparrow \alpha'$.

Proof. The proof is a straightforward induction on the definitions of $e \Downarrow v$ and $\alpha \lhd e$. We do only two cases; the remaining cases are similar.
Case (ap_ϕ). We must have $e' = f'(a')$ for some f', a'. Since $f \Downarrow \phi$ and $f \bowtie f'$, by part 1 of the induction hypothesis we cannot have $f' \Downarrow \lambda x.\ b'$ for any b', and so $f'(a') \Downarrow v'$ must be derived by an instance of rule (ap_ϕ). By part 1 of the induction hypothesis, $\phi \uparrow \phi'$, and by part 2, $\alpha \uparrow \alpha'$. By definition of $\phi \uparrow \phi'$, $\beta \uparrow \beta'$.
Case (ap_ξ). Proceeding as in the previous case, we have $e' = f' \cdot a'$ and $f' \cdot a' \Downarrow \beta'$ via (ap_ξ). By the induction hypothesis, $\xi \uparrow \xi'$, so there exist $\alpha \in \xi$ and $\alpha' \in \xi'$ such that $\alpha \uparrow \alpha'$. We have $f(\alpha) \bowtie f'(\alpha')$, so by the induction hypothesis $\beta \uparrow \beta'$.

Theorem 9. (Coherence.) *Suppose* $e \in T_0$.

1. *If* $\gamma_1 \lhd e$ *and* $\gamma_2 \lhd e$ *then* $\gamma_1 = \gamma_2$.
2. *For all* $\gamma \in V$ *and* $\alpha_1, \alpha_2 \in \gamma$, *if* $\hat{\alpha_1} \lhd e$ *and* $\hat{\alpha_2} \lhd e$ *then* $\alpha_1 = \alpha_2$.

Proof. For part 2, if $\alpha \lhd e$ and $\alpha' \lhd e$ for $\alpha, \alpha' \in \gamma$, then by Lemma 8, $\alpha \uparrow \alpha'$, and so $\alpha = \alpha'$ by the definition of V. The proof of part 1 is similar.

3 Nuprl

This section shows how to apply the semantic ideas of the previous section to a variant of Nuprl's type theory.

Nuprl has a large number of built-in type constructors and a very large number of inference rules (close to 100), so a complete account here is impossible. However, the semantics is sufficient simple that an interested reader would not have too much difficulty in verifying all the rules given in the Nuprl book [5], given the definitions and examples in this section, and assuming the results of Section 2.

To give the semantics for the type theory, we first need to extend the operational semantics to include evaluation rules for all the type constructors. It is easy to show that all the results of the previous section hold for this extension. We only give a few examples of such rules. Obvious variations work for the other type constructors.

To account for all of Nuprl, we have to make V sufficiently large. Nuprl has a hierarchy of universes $U_1, U_2, \ldots$ of types. Each U_i has to be closed under type constructors such as generalized cartesian product. This means that the set-theoretic meaning of each U_i has to be closed under the corresponding set constructors. This requires the use of *inaccessible cardinals*. These are defined in most set theory texts, and the reason for their use in this context is explained further in [11]. We choose the ordinal σ_0 in the definition of W to be the limit of a countable sequence $\tau_1 < \tau_2 \ldots$ of inaccessible cardinals. For each $i \geq 1$, let $\gamma^i = V \cap Z_{\tau_i}$. We add evaluation rules $U_i \Downarrow \widehat{\gamma^i}$ for each $i \geq 1$.

We give a rule for Nuprl's generalized cartesian product $x : A \rightarrow B$ as follows. If $\gamma \in V$ and, for each $\alpha \in \gamma$, $\gamma_\alpha \in V$, then let $\Pi \alpha \in \gamma\, .\, \gamma_\alpha$ denote the set of all

$\phi \in V$ such that the domain of ϕ is γ, and for each $(\alpha, \beta) \in \phi$, $\beta \in \gamma_\alpha$. Note that $(\Pi\, \alpha \in \gamma\, .\, \gamma_\alpha) \in V$. The evaluation rule is

$$\frac{A \Downarrow \hat{\gamma} \quad \forall\, \alpha \in \gamma.\ B[\hat{\alpha}/x] \Downarrow \widehat{\gamma_\alpha}}{x : A \rightarrow B \ \Downarrow\ i[\Pi\, \alpha \in \gamma\, .\, \gamma_\alpha]}$$

Nuprl has an equality type, similar to Martin-Löf's "I" type, which represents the proposition that two elements of the type are equal. Let γ_{true} be some one-element set, and let γ_{false} be the empty set. The rules for the equality type are as follows.

$$\frac{A \Downarrow \hat{\gamma} \quad \alpha \in \gamma \quad \alpha \lhd a_1 \quad \alpha \lhd a_2}{(a_1 = a_2 \in A) \Downarrow \widehat{\gamma_{true}}} \qquad \frac{A \Downarrow \hat{\gamma} \quad \alpha \neq \beta \in \gamma \quad \alpha \lhd a_1 \quad \beta \lhd a_2}{(a_1 = a_2 \in A) \Downarrow \widehat{\gamma_{false}}.}$$

Finally, we give a rule for the quotient type. If X is an equivalence relation over γ, then let $\gamma//X$ be the set of equivalence classes of X. In the rule below, let Q stand for $\{\, (\alpha, \beta) \in \gamma \times \gamma \mid \gamma_{\alpha,\beta} \neq \emptyset \,\}$.

$$\frac{A \Downarrow \gamma \quad \forall\, \alpha, \beta \in \gamma.\ E[\hat{\alpha}, \hat{\beta}/x, y] \Downarrow \widehat{\gamma_{\alpha,\beta}} \quad Q \textit{ is an equivalence relation}}{(x, y) : A//E \ \Downarrow\ i[\gamma//Q].}$$

Having added the evaluation rules for all the type constructors, we can give the semantics of Nuprl's type system.

Definition 10. A closed term e is a *type* if there is a $\gamma \in V$ such that $e \Downarrow \hat{\gamma}$. Define $M_*[e] = \gamma$.

M_* is well-defined by Theorem 9 and the fact that $e \Downarrow \gamma$ implies $\gamma \lhd e$.

Definition 11. Let e and a be closed terms. Define $a \in e$ if e is a type and there exists $\alpha \in M_*[e]$ such that $\alpha \lhd a$. In the case α exists, define $M_e[a] = \alpha$.

M_e is well-defined by Theorem 9. We will write $M_\gamma[a]$ for $M_{\hat{\gamma}}[a]$. Note that if $e \in U_i$ then e is a type and $M_*[e] \in \gamma^i$.

The preceding definition gives the core idea of the semantics. With a type is associated a set γ, and the members of the type are all terms e which are approximated by some member of γ.

It is now easy to give the semantics of Nuprl sequents. Nuprl sequents have the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash t \in T$$

where $t, T, A_1, \ldots, A_n$ are terms and for all i, $1 \leq i \leq n$, all of the free variables of t and T are among $x_1, \ldots, x_n$, and all of the free variables of each A_i are among $x_1, \ldots, x_{i-1}$. A *closing substitution* for the sequent is a substitution which has domain $x_1, \ldots, x_n$ and whose values are all closed terms.

Definition 12. The sequent

$$x_1 : A_1, \ldots, x_n : A_n \vdash t \in T$$

is *true* if $\sigma(t) \in \sigma(T)$ for all closing substitutions σ such that for all i, $1 \leq i \leq n$, $\sigma(x_i) \in \sigma(A_i)$.

Soundness of an inference rule is now defined as usual: the conclusion is true whenever the premises are.

Theorem 13. *The rules of Nuprl, with suitable modifications to the quotient and extensionality rules, are sound.*

We do not give the proof here, but just give a few representative cases. We also simplify the rules somewhat, for example by dealing only with the hypotheses that are introduced or analyzed by a rule (as opposed to the common prefix of hypotheses shared by the conclusion and premises).

We first do the "function intro" rule.

$$\frac{\vdash A \in U_i \quad x{:}A \vdash b \in B}{\vdash \lambda x.\, b \ \in \ x : A \rightarrow B}$$

By the first premise, there is a γ such that $M_*[A] = \gamma$. By the second premise, for each $\alpha \in \gamma$, $B[\hat{\alpha}/x]$ is a type, so for some γ_α, $M_*[B[\hat{\alpha}/x]] = \gamma_\alpha$. By the evaluation rule for the product type,

$$M_*[x : A \rightarrow B] \ = \ \Pi \alpha \in \gamma \,.\, \gamma_\alpha .$$

By the second premise, $b[\hat{\alpha}/x] \in B[\hat{\alpha}/x]$ for all $\alpha \in \gamma$. Let $\phi \in \Pi \alpha \in \gamma \,.\, \gamma_\alpha$ encode the mapping $\alpha \mapsto M_{\gamma_\alpha}[b[\hat{\alpha}/x]]$. To complete the verification of the rule, we only need to show that $\phi \leq \lambda x.\, b$, and this is immediate.

Next we do the "function elim" rule.

$$\frac{\vdash a \in A \quad \vdash f \in \ \ x : A \rightarrow B}{\vdash f(a) \in B[a/x]}$$

Let $M_A[a] = \alpha$ and $M_{x:A \rightarrow B}[f] = \phi$. Let β be such that $(\alpha, \beta) \in \phi$. By the evaluation rule for product, $B[\hat{\alpha}/x] \Downarrow \gamma$ for some γ such that $\beta \in \gamma$, so, using Theorem 6, $\gamma \leq B[\hat{\alpha}/x] \leq B[a/x]$ and hence $M_*[B[a/x]] = \gamma$. Since $\phi \leq f$, we have $\hat{\beta} \leq f(\hat{\alpha}) \leq f(a)$, so $f(a) \in B[a/x]$.

Some of the rules for Nuprl's quotient type need to be changed. We sketch these changes below. Verification of the rules is straightforward, and is omitted here. Let Q be $(x, y) : A//E$. The new "introduction" rule adds the constructor $[\cdot]$.

$$\frac{\vdash Q \in U_i \quad \vdash a \in A}{\vdash [a] \in Q}$$

The new "elimination" rule adds the non-canonical form for quotients.

$$\frac{\vdash e \in Q \quad x{:}A, y{:}A, z{:}E \vdash c_{true} \in (f(x) = f(y) \in T)}{\vdash f \cdot e \in T}$$

where c_{true} is the unique member of γ_{true}. In the second premise, the assumption $z{:}E$ merely asserts that E is true; the variable z is not allowed to occur free in f or T. Also, there is no need for a premise giving a type to f.

One other rule for quotients also needs to be changed. The changes are similar to the "intro" rule. Also, Nuprl's "direct computation" rule, which formalizes symbolic computation, must be updated to include the equivalence $f \cdot [e] \sim f(e)$.

4 HOL

In this section we first sketch how to use the semantics we have developed to justify an embedding of HOL. We then explain why this embedding is effective.

In order to embed HOL, we need to add an operator to Nuprl to represent HOL's "select" operator. The evaluation rule is as follows. Note that nonemptiness of a type is taken to represent truth of the corresponding proposition.

$$\frac{T \Downarrow \hat{\gamma} \quad \forall \alpha \in \gamma.\ P[\hat{\alpha}/x] \Downarrow \widehat{\gamma_\alpha} \quad \alpha_0 \in \gamma \text{ of minimum rank such that } \gamma_{\alpha_0} \neq \emptyset}{@x \in T.\ P \quad \Downarrow \quad \widehat{\alpha_0}}$$

The base logic of HOL is a polymorphic version of the simply typed λ-calculus with two base types: *bool*, for the booleans, and *ind*, representing an infinite set. There are three constants

$$\begin{aligned} = &: \text{'}a \to \text{'}a \to bool \\ ==> &: bool \to bool \to bool \\ @ &: (\text{'}a \to bool) \to \text{'}a \end{aligned}$$

for equality, implication, and "select", respectively. The *'a* is a type variable. The semantics of HOL is the standard set-theoretic one where function type is given the usual set-theoretic meaning.

Given Nuprl's select operator, it is trivial to give definitions in Nuprl for the base types and the constants. *ind* is defined to be Nuprl's type of natural numbers, and HOL's function type is interpreted as Nuprl's function type. We can prove in Nuprl that each of the (defined) constants has the appropriate type. For example, we can prove

$$\forall \text{'}a \in HOL\_U.\ @ \in (\text{'}a \to bool) \to \text{'}a$$

where HOL_U is the type of all members of the type universe U_1 that are nonempty, and *bool* is defined as the subtype $\{ x \in Z \mid x = 0 \vee x = 1 \}$ of Nuprl's type Z of integers. In general, all HOL theorems and axioms have implicit outermost quantifiers for their type variables. These quantifiers are made explicit in the Nuprl interpretation.

Extensions to the HOL logic are made by creating *theories*. A theory consists of a set of new type constructors, and set of new constants with their types, some axioms involving the new constants and types, and a set of theorems proved using the axioms. To import this theory into Nuprl, we proceed as we did for the base logic: we find Nuprl objects to use in place of the new types and constants (these can usually be computed automatically from the theory's definitions), prove in Nuprl that the constants have their assigned types, and then prove that the axioms all hold. Once we have done this, the new semantics of Nuprl tells us that the set theoretic meaning of the Nuprl objects we introduced can be directly used to give a model of the HOL theory. The theorems are valid in this model, hence their translations into Nuprl are true.

The use of Nuprl objects to "instantiate" HOL theories is one reason why this embedding is effective. For example, we instantiate the HOL theories for the integers with Nuprl's integer type and Nuprl's built-in operations over the integers. All the HOL theorems about the HOL integers then become applicable to Nuprl integers.

A major concern here is the constructivity of Nuprl proofs. HOL formulas, when imported into Nuprl, have type *bool*. Nuprl's encoding of logic uses propositions-as-types, so, for example, Nuprl's universal quantification is represented using the generalized cartesian product type. We can prove, however, that the two ways of encoding logic are equivalent. Thus, for each HOL theorem imported, we can derive a version of the theorem that uses Nuprl's logical connectives. However, the proof of this derivation is non-constructive, and so the program synthesized from one of these new theorems might mention the non-computable select operator @. If such a theorem is cited in another Nuprl proof, the program there might also be uncomputable.

The reason this is not a problem is because equalities in Nuprl have no computational content. So, for example, if a universally quantified equation is proved, then the program extracted from the proof is simply a constant function. This has two main consequences. First, if we are proving an equation (possibly under assumptions) in Nuprl, we can safely use any HOL theorem whatsoever. Second, no matter what we are proving, it is always safe to use HOL facts, such as universally quantified equations, that have no computational content. Fortunately, the vast majority of HOL theorems fit this category, and the vast majority of the work in proving any theorem about software involves computationally trivial facts (mostly equations and inequations). Most of the work in Nuprl proofs is done by term rewriting. All the programs that apply term rewriting can safely use any HOL theorem.

It is easy to modify the system to ensure that non-computable "programs" are not inadvertently extracted from proofs. For example, we can add a bit to each proof node, where a true bit means that the extracted program of the subproof rooted at the node must not contain the select operator. The user sets the bit at the root, and the system computes the bit when the proof is extended by refinement, setting it to false when the node being refined has a conclusion which is computationally trivial, and simply propagating it otherwise. Inference steps may not mention the select operator, or use lemmas whose top bit is false, if the bit at the node being refined is true.

Details on the actual HOL embedding and some of its practical applications will be given in a forthcoming paper.

5 Related Work, Discussion

In [4], Breazu-Tannen and Subrahmanyam give a logic for reasoning about programs using structural recursion over data types formed from constructors subject to some equations. Their idea, to make the meaning of a definition by structural recursion – if it does not respect the equations, is somewhat similar to

our treatment of $f \cdot t$, which tests to see if f respects equality as given by the equivalence class t.

It is straightforward to give a classical interpretation justifying the select operator for typed variants of Martin-Löf's type theory. Such an interpretation is given by Dybjer in [6]. Because these variants are based on a typed language, they do not enjoy some of the useful features of Nuprl, such as direct reasoning about the programming language used for realizers extracted from proofs, writing programs in a conventional style (e.g. general recursion), genericity with respect to programming language (e.g. using ML instead) and partial functions.

The idea of adding function oracles ϕ to an untyped programming language comes from our paper [11]. The semantics we gave there might give a model of Nuprl in which the select operator is definable, but we would have had to use the complicated PER semantics along the lines discussed earlier (the author gave up on doing this in disgust), and the embedding of HOL would be much harder to justify because of the PER/set mismatch in the respective semantics.

The idea for the proof of Theorem 6 first appeared in [10]. An expanded treatment of the proof method is in [11].

In [13] we gave a type theory that contained ZF set theory. Although that theory is in some respects similar to Nuprl, the programming language in the theory is typed in an essential way and hence the semantics cannot be applied to Nuprl. Also, there are no polymorphic equivalence classes in that work, and the semantics is much more complicated than the present one.

The new semantics justifies some useful extensions to Nuprl. It justifies extensional equality of types: two types are equal if and only if they have the same set of members. Currently, Nuprl's type equality is taken to be intensional, or structural, for technical reasons having to do with collapsing Martin-Löf's four forms of judgment to one, although there are no rules that exploit this. Other extensions include: the law of the excluded middle (an immediate consequence of the introduction of the @ operator), a power set constructor, and some impredicative constructs.

References

1. *Part IIIA: SCI Coherence Overview, 1995.* Unapproved draft IEEE-P1596-05Nov90-doc197-iii.
2. *Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science. Springer, 1995.
3. S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, 1987.
4. V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Automata, Languages and Programming: 18^{th} International Colloquium*, Lecture Notes in Computer Science, pages 60–75. Springer-Verlag, 1991.
5. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

6. P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Proceedings of the B.R.A. Workshop on Logical Frameworks*, Sophia-Antipolis, France, June 1990.
7. A. Gordon. *Functional Programming and Input/Output.* Cambrige University Press, 1994.
8. M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
9. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, UK, 1993.
10. D. J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, June 1989.
11. D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.
12. D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 1996. To appear.
13. D. J. Howe and S. D. Stoller. An operational approach to combining classical set theory and functional programming languages. In *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
14. P. B. Jackson. Exploring abstract algebra in constructive type theory. In A. Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artifical Intelligence. Springer, June 1994.
15. B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1993.
16. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
17. T. Melham. The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3(1–2):7–24, August 1993.
18. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
19. E. Ritter and A. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer, 1995.
20. M. van der Voort. Introducing well-founded function definitions in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume A-20 of *IFIP Transactions*, pages 117–131. North-Holland, 1993.

A Proof of Precongruence

The proof of Theorem 6 in large part a straightforward extension of the method of [10, 12]. One of the main differences is that we use Lemma 14 for the cases for rules (ap_ϕ) and (ap_ξ).

Lemma 14. *If $\alpha \lhd e[\hat{\beta}/x]$ and $\beta \lhd e'$ then $\alpha \lhd e[e'/x]$.*

Lemma 14, while highly plausible, is in fact rather difficult to prove. The obvious inductive argument over the definition of $\lhd$ fails because of rules (ap_ϕ) and (ap_ξ). Our proof is rather complicated, and is omitted because of lack of space. All the complication resides in the well-founded relation that the inductive proof is based on.

We now sketch the remainder of the proof. We continue to omit any mention of operators not treated in Section 2. The key to the rest of the proof is the following definition.

Definition 15. Define the *precongruence candidate*, a binary relation $\widehat{\leq}$ on terms, by induction on the size of its first argument: $e \mathrel{\widehat{\leq}} e'$ if there exists e'' such that $e \mathrel{\langle\widehat{\leq}\rangle} e''$ and $e'' \leq^\circ e'$.

It is straightforward to verify that the precongruence candidate has the following properties: it is reflexive; it is *operator respecting* in the sense that if $e \mathrel{\widehat{\leq}} e'$ then $C[e] \mathrel{\widehat{\leq}} C[e']$ for all contexts $C[\cdot]$; if $e_1 \mathrel{\widehat{\leq}} e_2 \leq e_3$ then $e_1 \mathrel{\widehat{\leq}} e_3$; if $v \mathrel{\widehat{\leq}} v'$ then $v \mathrel{[\widehat{\leq}]} v'$; and if $e \leq^\circ e'$ then $e \mathrel{\widehat{\leq}} e'$. We can also show that $\widehat{\leq}$ is substitutive: if $e \mathrel{\widehat{\leq}} e'$ and $b \mathrel{\widehat{\leq}} b'$ then $b[e/x] \mathrel{\widehat{\leq}} b'[e'/x]$.

The following key lemma can be easily proved using co-induction on the definition of $\leq$.

Lemma 16. *Suppose that for all closed e, e' and v such that $e \Downarrow v$ and $e \mathrel{\widehat{\leq}} e'$, there exists v' such that $e' \Downarrow v'$ and $v \mathrel{\widehat{\leq}} v'$. Then $\leq$ is a precongruence.*

We can now prove Theorem 6.

Proof. By the preceding lemma, it suffices to prove by induction on the definitions of $\Downarrow$ and $\lhd$ that

1. If $e \Downarrow v$ and $e \mathrel{\widehat{\leq}} e'$ then $e' \Downarrow v'$ with $v \mathrel{\widehat{\leq}} v'$.
2. If $\alpha \lhd e$ and $e \mathrel{\widehat{\leq}} e'$ then $\alpha \lhd e'$.

We do a case analysis on rules. In each case, we use the following fact. Let $1'$ be property 1 above with $e \mathrel{\widehat{\leq}} e'$ replaced by $e \mathrel{\langle\widehat{\leq}\rangle} e'$. It is straightforward to show that for any particular e, e', v and v', $1'$ implies 1. An analogous statement holds for property 2.

We only consider a few representative cases. The remaining cases are either very similar or trivial.

Case (ap_ϕ). Suppose $f(a) \mathrel{\langle\widehat{\leq}\rangle} f'(a')$. $f' \Downarrow v'$ with $\phi \mathrel{\widehat{\leq}} v'$. By definition of $\widehat{\leq}$, $\phi \leq v'$. Suppose $v' = \phi$. By part 2 of the induction hypothesis, $\alpha \lhd a'$, so $f'(a') \Downarrow \beta$. Suppose $v' = \lambda x.\ b$. Since $\beta \leq b'[\alpha/x]$, by Lemma 14 we have $\beta \leq b'[a'/x]$, so $b'[a'/x] \Downarrow v''$ with $\beta \leq v''$, and this implies $\beta \mathrel{\widehat{\leq}} v''$.

Case (ap_λ). Suppose $f(a) \mathrel{\langle\widehat{\leq}\rangle} f'(a')$. $f \Downarrow \lambda x.\ b$ so by the induction hypothesis there exists b' such that $f' \Downarrow \lambda x.\ b'$ and $b \mathrel{\widehat{\leq}} b'$. $b[a/x] \mathrel{\widehat{\leq}} b'[a'/x]$, so by the induction hypothesis $v \mathrel{\widehat{\leq}} v'$.

Case $(\phi\lhd)$. Suppose $\lambda x.\ b \mathrel{\langle\widehat{\leq}\rangle} e'$. Then $e' = \lambda x.\ b'$ where $b \mathrel{\widehat{\leq}} b'$. $b[\hat{\alpha}/x] \mathrel{\widehat{\leq}} b'[\hat{\alpha}/x]$, so $\beta \lhd b'[\hat{\alpha}/x]$ and $\phi \lhd \lambda x.\ b$.